UNIVERSIDAD AUTÓNOMA DE MADRID

Escuela Politécnica Superior

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

EXTRAPOLACIÓN DE IMÁGENES MEDIANTE PROCESOS GAUSSIANOS

Pablo Moreno Martín

Tutor: Daniel Hernández Lobato

Mayo 2016

Extrapolación de Imágenes Mediante Procesos Gaussianos

Autor: Pablo Moreno Martín
Tutor: Daniel Hernández Lobato

..

Ing. Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Mayo 2016

RESUMEN

Resumen La capacidad de descubrir y aprender patrones de diferentes conjuntos de datos es uno de los objetivos fundamentales del Aprendizaje Automático. Dentro de esta inmensa área de conocimiento, los Procesos Gaussianos (GPs) suponen una potente herramienta para tratar tanto para problemas de regresión como de clasificación. Al tratarse de un método no paramétrico basado en kernels, permite el aprendizaje de todo tipo de patrones en diferentes conjuntos de datos siempre y cuando compartan ciertas propiedades con los kernel escogidos. En este caso utilizaremos dichos Procesos Gausianos para la extrapolación de imágenes incompletas permitiendo así su reconstrucción.

Una de las principales características de un GP es el kernel escogido para su implementación, ya que los resultados obtenidos heredarán propiedades muy relevantes de estos kernel. Para poder aprender de forma correcta la estructura de las imágenes necesitaremos utilizar kernels con una capacidad muy grande de aprender patrones muy variados, es por ello que utilizamos para esta aplicación concreta los conocidos como "Kernel Superexpresivos" que nacen como combinaciones de kernels más simples y que permiten al modelo aprender patrones muy dispares y complejos.

Uno de los principales problemas de los Procesos Gaussianos son sus grandes exigencias computacionales, lo que los hace inviables en problemas donde los conjuntos de datos son relativamente grandes, como pasa en el caso de las imágenes. Sin embargo en este caso, al igual que ocurre en otros problemas similares, los conjuntos de datos están estructurados (las imágenes pueden verse como valores en una rejilla). Aquí estudiamos cómo podemos aprovechar dicha estructura subyacente en los datos para poder reducir drásticamente sus costes operacionales y hacer viable el uso de procesos Gaussianos en dichos conjuntos de datos.

Además de una exposición del modelo teórico utilizado en los GPs, también se muestran varias pruebas realizadas y reconstrucciones de imágenes completas a través del algoritmo, donde podrán verse los resultados del mismo para la tarea llevada a cabo. Del mismo modo mostraremos una comparación realizada con problemas en los que no se explota la estructura de los conjuntos de datos, para poder diferir las mejoras computacionales conseguidas por nuestro modelo.

Para la implementación de los modelos estudiados se ha utilizado el lenguaje de programación Julia, un lenguaje de programación relativamente nuevo con una sintaxis

sencilla, diseñado para la utilización en programas con aplicaciones de cálculo numérico y aprendizaje automático.

Palabras clave Procesos Gaussianos, kernel, extrapolación, regresión, Julia, aprendizaje automático

Abstract

Abstract The capacity to discover and learn patterns from different data sets is one of the main objetives of Machine Learning. Within this huge knowledge area, Gaussian Processes (GPs) are a powerful instrument to address two problems: classification and regression. Due to the fact that GPs are a non-parametric model based on kernels, they allow learning every kind of pattern in very different data sets with propieties that are similar to those that has the choosen kernel. In this particular case, we will use the Gaussian Processes to extrapolate images that are incomplete in order to reconstruct them.

One of the main characteristics of a GP is the kernel chosen for its implementation because the results that we get will inherit propierties from these kernels that are very relevant. In order to learn correctly the structure of the images, we will need to use kernels that have an enormous capacity of learning a wide variety of patterns, and because of this, we will use for this particular aplication the "Superexpressive Kernels" which originate from combinations of simpler kernel and that allow our model to learn complex patterns.

One of the problems of Gaussian Processes is their high computational cost, that makes them impossible to use in those problems where the amount of data that we have to analyse is relatively large, as it happens in the case of images. Nevertheless, in the case of images, as it happens in others similar problems, the data set has some structure (we can see an image as a set of values on a grid), and here we study how can we use this particular structure in the data set to reduce the computational cost of GPs and make it possible to use them in such big data sets.

In addition to an exposition of the theoretical model that is used under a GP, we will show some tests that we have done with GPs and some reconstruction of images using the algorithm described, where we will be able to see it results. We will compare the performance that we get using our model in compare with that of general GPs.

For the implementation of the models that are presented int this bachelor work, we have used the programming language Julia, which is a new language with an easy syntax, designed to solve problems in the areas of numerical analysis and machine learning.

Keywords Gaussian Processes, kernel, extrapolation, regression, Julia, Machine Learning

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN Y MOTIVACIÓN

En este primer capítulo se detalla la motivación que ha llevado a la realización de este trabajo, así como los objetivos generales y específicos perseguidos en su realización. Además presentaremos al final de este capítulo la estructura global del presente documento.

1.1 Motivación

La capacidad de aprender y extrapolar patrones de conjuntos de datos siempre ha sido uno de los objetivos principales del Aprendizaje Automático y más concretamente del *Aprendizaje Supervisado*. Entre las muchas clasificaciones que se pueden hacer de los modelos de regresión y extrapolación de aprendizaje automático, una de las más comunes es separar los modelos en *modelos paramétricos* y *modelos no paramétricos*.

En los primeros, las distribuciones estadísticas de los conjuntos de datos pueden ser dadas a priori a través del uso de, valga la redundancia, parámetros que contienen, en cierta medida, información acerca del problema a resolver en general y del conjunto de datos utilizado en particular. Sin embargo para la utilización de estos modelos es necesario un conocimiento bastante específico acerca de las características del problema en concreto; ya que dichos parámetros son fijados previamente por la persona que esté generando el modelo.

En los segundos, sin embargo, el número de parámetros no esta fijo y crece con el número de ejemplos de train. Es decir, la complejidad del modelo crece con el número de datos. Es en estos últimos en los que se centra el presente trabajo, más concretamente en los **Procesos Gaussianos (GPs)**.

Un Proceso Gaussiano (GP) es un modelo de regresión que consiste en un proceso estocástico continuo que asigna a cada punto una distribución de probabilidad. Para la asignación de esta distribución se toman ciertas asunciones acerca del tipo y las características generales de dicha distribución que son expresadas en el GP mediante el Kernel utilizado. Aunque más adelante veremos características específicas de estos kernels, es importante mencionar que nuestra función de distribución posteriori aprendida con los datos, heredará ciertas propiedades de estos Kernel(suavidad, perodicidad, etc.). Es por ello que la elección de un Kernel apropiado es extremadamente importante a la hora de definir el GP. El hecho de utilizar dichas funciones hace que podamos clasificar los GPs dentro de los *Métodos Kernel*.

Dentro del campo del *Aprendizaje Supervisado*, el problema de aprender la estructura inherente a una imagen siempre ha sido un reto recurrente; bien para la comparación de imágenes o para la reconstrucción o tratamiento de las mismas. En este trabajo nos centraremos en la **reconstrucción de imágenes**, ya sea de áreas que se ven borrosas por exceso de ruido o simplemente de áreas que faltan en la imagen. Este problema de la reconstrucción de imágenes ha permanecido sin solución durante muchos años, salvo aquellos casos en las que se completaban las áreas que faltaban con copias de otras zonas de la misma imagen y en este caso proponemos una solución basada en el aprendizaje de la estructura intrínseca de las imágenes.

1.2 Objetivos

El objetivo general de este trabajo es hacer un estudio teórico de de los GP en general y de su uso para el problema de la extrapolación y reconstrucción de imágenes en particular.

Uno de los principales problemas de los GPs es su alto coste computacional. La cantidad de operaciones necesaria para poder realizar el aprendizaje del modelo es extremadamente alta, lo que hace que los GP sean inviables en problemas donde los conjuntos de datos tengan un tamaño excesivamente grande, como puede ser el caso de las imágenes, pues una imagen de 840x600 pixels tendría un total de 480000 datos que, aunque no son demasiados en comparación con otros problemas del *Aprendizaje Automático*, es una cantidad más que considerable.

Sin embargo en el caso de las imágenes, al igual que pasa en otros problemas, existe cierta estructura en los datos que nos permite aprovecharnos de ella y realizar la implementación de los GP en un tiempo varios ordenes de magnitud menor al que cabría esperar [6]. Detalles cuantitativos más precisos sobre el ahorro conseguido se darán más adelante en el presente documento.

La estructura mencionada sobre los datos en las imágenes es el hecho de que las imágenes puedan representarse en una rejilla, donde las posiciones dentro de dicha rejilla pueden darse como un producto cartesiano entre los dos ejes de coordenadas. Más adelante veremos como explotar esta estructura de los datos en las imágenes.

Otro de los problemas de los GP, como ya se ha mencionado anteriormente, es la necesidad de la elección del Kernel, pues tiene que tener la capacidad y las propiedades necesarias para aprender de forma correcta la estructura de los datos. En este caso utilizaremos los conocidos como **Kernel Superexpresivos** [10], de los que hablaremos con detalle más adelante, que permiten aprender y adaptarse a estructuras de datos muy dispares y complejas mediante la combinación de Kernel más simples.

Toda la implementación de los modelos y los algoritmos necesarios será llevada a cabo en el novedoso lenguaje de programación **Julia**; que presentaremos un poco más adelante y que ofrece un rendimiento cercano al de *C* con una sintaxis simple, similar a la de otros lenguajes como *Matlab* o *R* por lo que su uso está orientado a áreas como el Aprendizaje Automático o el Análisis de datos.

1.3 Estructura del documento

En los distintos apartados del documento se detallan todos los pasos y consideraciones realizadas para poder construir el algoritmo completo de extrapolación de imágenes mediante GPs.

De esta forma, en el capítulo 2 explicaremos el modelo teórico de los GPs en general, dentro del marco del aprendizaje supervisado. En este capítulo veremos también ejemplos de utilización de los GPs para problemas de regresión simples.

En el capítulo 3 estudiaremos cómo podemos adaptar los GPs genéricos explicados anteriormente al problema de extrapolación de imágenes en general, y cómo mejorar enormemente su rendimiento para este tipo de problemas en particular. Este capítulo estará divido en dos bloques: en el primero se tratarán imágenes que estén completas y de las que se quiera extrapolar información hacia el exterior y en el segundo veremos cómo tratar imágenes que no estén completas y de las que se quiera extrapolar zonas del interior de la imagen.

Después de estudiar todo el modelo teórico del algoritmo a utilizar, en el capítulo 4 veremos ejemplos de utilización del algoritmo para la reconstrucción de imágenes reales y algunos casos de extrapolación en ejemplos simplificados, en este capítulo detallaremos todas las consideraciones adicionales realizadas sobre el modelo para su correcto funcionamiento.

Por último, en el capítulo 5 expondremos las conclusiones obtenidas tras la realización del trabajo.

ESTADO DEL ARTE

2

2.1 Introducción al Aprendizaje Supervisado

El **Aprendizaje Automático** es uno de los campos de estudio más relevantes dentro de la *Inteligencia Artifical*. En la conocida como *"Era de los Datos"*, el Aprendizaje Automático representa todo el conjunto de técnicas, algoritmos y modelos matemáticos que sirven para analizarlos, sacar información sobre ellos que podría ser utilizada para hacer nuevas predicciones.

Una de las clasificaciones más influyentes dentro de los algoritmos de Aprendizaje Automático es la clasificación entre **Aprendizaje No Supervisado** y **Aprendizaje Supervisado**.

El primer tipo simplemente tiene como objetivo extraer información de los datos que puede resultar potencialmente interesante sin ningún objetivo en particular. Este tipo de información puede incluir patrones intrínsecos que siguen los datos, relaciones entre los mismos, etc. Al no tener un objetivo marcado, en este tipo de algoritmos no hay una métrica que pueda medir cómo de correctos son los resultados, por lo que, desde este punto de vista, los hace aún mas complejos si cabe.

Por otro lado, el segundo gran bloque dentro del Aprendizaje Automático es el Aprendizaje Supervisado, que es precisamente en el que nos centraremos en este trabajo. En este tipo de algoritmos el objetivo es establecer una relación entre ciertos datos de entrada a los que se les asigna ciertos datos de salida de la forma más precisa posible en base a unos ejemplos anteriormente dados que son los datos de entrenamiento.

Dentro del Aprendizaje Supervisado, en función del tipo de salidas que se pueden obtener, hay dos tipos. En primer lugar está la **Clasificación** que son aquellos problemas donde hay que asignar una "clase" de salida para ciertos datos de entrada de entre una cantidad finita de clases. De ahí que el propio nombre sea "Clasificación" pues lo que hace es clasificar en cierta manera los datos.

El otro tipo de Aprendizaje Supervisado es el conocido como **Regresión**, y es en éste en el que se pueden incluir los algoritmos descritos en el presente trabajo (aunque también pueden adaptarse para que se puedan usar para Clasificación como pasa con muchos de los algoritmos del Aprendizaje Automático). En el caso de la Regresión, el objetivo es asignar unos valores reales de salida para los datos de entrada que pueden estar dentro de un conjunto infinito.

Así, en este caso, la regresión que haremos será la de dados ciertos valores de entrada (posiciones de los pixeles en una imagen) le asignaremos un valor real de salida, que corresponderá con el color de la imagen en dicha posición.

2.2 Procesos Gaussianos

2.2.1 Conceptos previos

Antes de definir lo que son los GPs, considero que es necesario tener una pequeña base teórica probabilística para poder entender los fundamentos que hay detrás de los GP en general y de su uso para la regresión en particular.

Definición 1 (Variable aleatoria)**.** *Una Variable aleatoria (VA) X es una función real definida en el espacio de probabilidad $(\Omega, \mathcal{A}, P)$ asociado a un experimento aleatorio .*

$$X : \Omega \to \mathbb{R} \tag{2.1}$$

que asocia a cada suceso elemental un número real.

Definición 2 (Variable aleatoria continua)**.** *Una VA X es una variable aleatoria continua si toma valores en un conjunto de partida no numerable.*

Definición 3 (Función de Distribución)**.** *En teoría de la probabilidad una función de distribución acumulada describe la probabilidad de que una variable aleatoria real X sujeta a cierta ley de distribución de probabilidad se sitúe en la zona de valores menores o iguales a x. Se representa como:*

$$F(x) = \mathbb{P}(X \leq x) \tag{2.2}$$

Definición 4 (Función de Densidad)**.** *En la teoría de la probabilidad, la función de densidad de probabilidad, función de densidad, o, simplemente, densidad de una variable aleatoria continua describe la probabilidad relativa según la cual dicha variable aleatoria tomará determinado valor. Representa la derivada de la función de distribución. Sea $f(x)$ la función de densidad de cierta variable aleatoria entonces:*

$$\mathbb{P}[a \leq X \leq b] = \int_a^b f(x)\,\mathrm{d}x \tag{2.3}$$

Definición 5 (Distribución Normal)**.** *La función de densidad de una distribución normal viene dada por la siguiente expresión.*

$$\mathbb{P}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.4}$$

donde μ representa la media de la distribución y σ^2 representa la varianza. Una variable aleatoria que sigue esta distribución se denota como $X \sim \mathcal{N}(\mu, \sigma)$.

Definición 6 (Distribución normal multivariante (MVN))**.** *Un vector aleatorio no es más que un conjunto de variables aleatorias. Un vector aleatorio X es una normal p-dimensional con vector de medias μ y vector de covarianzas Σ si tiene una densidad dada por:*

$$f(x) = |\Sigma|^{\frac{-1}{2}} (2\pi)^{\frac{-p}{2}} \exp\left\{-\frac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu)\right\} \; x \in \mathbb{R}^p \tag{2.5}$$

Notación: $X \equiv N_p(\mu, \Sigma)$ *significa:* X *es normal p-dimensional con media* μ *y matriz de covarianzas* Σ*.*

Teorema 2.1 (Teorema de Bayes)**.** *Sea* $\{\mathcal{A}_1, ..., \mathcal{A}_n\}$ *un conjunto de sucesos mutuamente excluyentes y exhaustivos tales que la probabilidad de cada uno es distinta de 0. Sea* $\mathcal{B}$ *un suceso cualquiera del que se conocen las probabilidades* $\mathbb{P}(\mathcal{B}|\mathcal{A}_i)$*, entonces se cumple:*

$$\mathbb{P}(\mathcal{A}_i|\mathcal{B}) = \frac{\mathbb{P}(\mathcal{B}|\mathcal{A}_i)\mathbb{P}(\mathcal{A}_i)}{\mathbb{P}(\mathcal{B})} \tag{2.6}$$

donde $\mathbb{P}(\mathcal{A}_i)$ *son las probabilidades a priori,* $\mathbb{P}(\mathcal{B}|\mathcal{A}_i)$ *es la probabilidad de* $\mathcal{B}$ *condicionada a* $\mathcal{A}_i$ *y* $\mathbb{P}(\mathcal{A}_i|\mathcal{B})$ *son las probabilidades a posteriori.*

Los siguientes teoremas son los resultados fundamentales en torno a los cuales giran los GP, por lo que resultan extremadamente importantes. Se pueden encontrar más detalles sobre los mismos en [4, Capítulo 4.3].

Teorema 2.2 (Propiedad de Marginalización)**.** *Una propiedad muy relevante de las MVN es la siguiente. Si tengo dos vectores aleatorios con la siguiente distribución.*

$$X_1 = (x_1, ..., x_n) \sim \mathcal{N}(\mu_1, \Sigma_1),$$
$$X_2 = (x'_1, ..., x'_n) \sim \mathcal{N}(\mu_2, \Sigma_2)$$

Entonces se cumple la siguiente igualdad:

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right) \tag{2.7}$$

Lo que significa que si tengo dos vectores aleatorios X_1 *y* X_2 *con distribuciones normales, el conjunto de los dos vectores (su distribución conjunta) también sigue una distribución normal con la media y varianzas indicadas, dónde* Σ_{12} *y* Σ_{21} *indica la correlación de ambos vectores.*

Teorema 2.3 (Marginales y condicionales de una MVN)**.** *Sea* $x = (x_1, x_2)$ *una distribución normal conjunta con parámetros*

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \boldsymbol{\Sigma} = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}$$

con las distribuciones marginales dadas por

$$\mathbb{P}(x_1) = \mathcal{N}(\mu_1, \Sigma_{11}),$$
$$\mathbb{P}(x_2) = \mathcal{N}(\mu_2, \Sigma_{22})$$

Entonces las ***probabilidades condicionadas*** *vienen dadas por*

$$\mathbb{P}(x_1|x_2) = \mathcal{N}(\mu_{1|2}, \Sigma_{1|2}), \tag{2.8}$$
$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2)$$
$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$$

y de forma simétrica para la otra probabilidad condicionada.

Como ya se ha mencionado y podremos comprobar más adelante el teorema es muy importante. Para una demostración detallada consultar [4, Sección 4.3.4].

2.2.2 Introducción a los Procesos Gaussianos

En los modelos de regresión, observamos ciertos valores de entrada x_i que dan ciertos valores de salida y_i y asumimos que existe una función desconocida f tal que $y_i = f(x_i) + \epsilon_i$, donde ε_i representa el ruido. Nuestro objetivo es encontrar la distribución posterior sobre la función, es decir, $\mathbb{P}(f|X, y)$, y usar esta distribución para hacer predicciones sobre nuevos datos de entrada x_* resolviendo la siguiente ecuación

$$\mathbb{P}(y_*|x_*, X, y) = \int \mathbb{P}(y_*|f, x_*)\mathbb{P}(f|X, y)df \tag{2.9}$$

Un GP es simplemente un proceso estocástico continuo, es decir, una función de distribución distinta para cada punto, que tendrá una media $\mu(x)$ y una varianza $\Sigma(x)$.

Al tratarse de modelos no paramétricos, la complejidad del modelo para f crece con el número de datos. Lo que hace el GP es utilizar un prior; una función de distribución previa que se asume tiene características similares a la función que se quiere estimar, para construir, junto con los datos observados, una distribución sobre la función resultante o posterior $\mathbb{P}(f|X, y)$.

Para definir una distribución sobre una función simplemente tenemos que definir la distribución sobre un conjunto finito, arbitrario de puntos, es decir, si tenemos $x_1, ..., x_n$, el GP, asume que $\mathbb{P}(f(x_1), ..., f(x_n))$ es una distribución normal conjunta con una cierta media $\boldsymbol{\mu}(x)$ y una covarianza $\boldsymbol{\Sigma}(x)$ dada por $\Sigma_{ij} = \kappa(x_i, x_j)$ donde κ es un Kernel definido positivo, la principal idea detrás de todo esto es que si el Kernel designa que los valores x_i y x_j son similares, entonces los valores $f(x_i)$ y $f(x_j)$ se espera que sean similares. Es de aquí de donde se deduce la importancia de elegir un buen Kernel, pues él sera el encargado de discernir qué valores son similares entre ellos y cuáles no lo son y en qué medida [6].

2.2.3 Regresión con Procesos Gaussianos

Sea el prior de la regresión un GP denotado por

$$f(x) \sim GP(m(x), \kappa(x, x'), \tag{2.10}$$

$$m(x) = \mathbb{E}[f(x)]$$

$$\kappa(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))^T]$$

y sea $X = (x_1, ..., x_n)$ un conjunto finito de puntos, entonces el GP define una distribución normal conjunta dada por

$$p(f|X) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{2.11}$$

$$\Sigma_{ij} = \kappa(x_i, x_j)$$

$$\boldsymbol{\mu} = (m(x_1), ..., m(x_n))^T$$

normalmente tendremos que $m(x) = 0$ porque pasaremos los datos normalizados, haremos hincapié en ésto más adelante.

2.2.4 Predicción con Procesos Gaussianos

Supongamos que tenemos un conjunto datos de train $\mathcal{D} = (x_i, y_i), x = 1, ..., N$ con $y_i = f(x_i) + \varepsilon_i$ siendo ε_i el ruido, una variable aleatoria con distribución $\mathcal{N}(0, \sigma_y^2)$ que

se supondrá independiente para cada observación. En este caso la covarianza de dos observaciones vendrá dada por

$$cov[y_p, y_q] = \kappa(x_p, x_q) + \sigma_y^2 \delta_{pq}$$

siendo δ_{pq} una delta de kronecker que tomará valor 1 si $p = q$ y valor 0 en caso contrario. Es decir, para los datos de train tendremos

$$cov[\boldsymbol{y}|\boldsymbol{X}] = \boldsymbol{K} + \sigma_y^2 \boldsymbol{I}_N \tag{2.12}$$

Por lo tanto, para la distribución conjunta de los datos de train con las f_* de los datos de test, que representan los valores de la f para los datos de test sin ruido, que se quieren predecir tendremos

$$\begin{pmatrix} \boldsymbol{y} \\ f_* \end{pmatrix} \sim \mathcal{N}\left(0, \begin{pmatrix} \boldsymbol{K} + \sigma_y^2 \boldsymbol{I}_N & \boldsymbol{K}_* \\ \boldsymbol{K}_*^T & \boldsymbol{K}_{**} \end{pmatrix}\right) \tag{2.13}$$

donde $\boldsymbol{K}_*$ representa las covarianzas cruzadas entre los datos de test y los datos de train y $\boldsymbol{K}_{**}$ representa las covarianzas entre los datos de test. Asumimos que la media es 0 por cuestiones de normalización.

En este punto recordamos el teorema visto en el apartado 2.2.1 de las distribuciones condicionadas dadas por la ecuación 2.8 y en este caso tendremos

$$\begin{aligned} \mathbb{P}(f_*|\boldsymbol{X}_*, \boldsymbol{X}, \boldsymbol{y}) &= \mathcal{N}(\mu_*, \Sigma_*), \\ \mu_* &= \boldsymbol{K}_*^T (\boldsymbol{K} + \sigma_y^2 \boldsymbol{I}_N)^{-1} \boldsymbol{y} \\ \Sigma_* &= \boldsymbol{K}_{**} - \boldsymbol{K}_*^T (\boldsymbol{K} + \sigma_y^2 \boldsymbol{I}_N)^{-1} \boldsymbol{K}_* \end{aligned} \tag{2.14}$$

Es decir, ya tendremos una forma de asignar tanto la media como la varianza a los nuevos datos de test que queremos predecir.

2.2.5 Kernel

Gran parte de la calidad de la precisión en las predicciones de los GP depende del Kernel escogido. Intuitivamente el Kernel debería depender de $|x - x'|$ y ser por tanto invariable para traslaciones en el espacio o movimientos rigidos en general. No todos los Kernel cumplen esta propiedad, pero la mayoría de ellos la cumplen. Se les denomina *isotrópicos*. Es también necesario que los Kernel tengan una matriz semidifinida positiva, ya que si no, la matriz de covarianzas no estará bien definida. Más detalles sobre las propiedades de los Kernel pueden encontrarse en [6, Capítulo 4].

El verdadero potencial de los Kernel reside en que representan un producto escalar entre espacios vectoriales de funciones con dimensión arbitrariamente grande, es decir, podemos ver el Kernel de la siguiente manera: $\kappa(x, x') = \phi(x)^T \phi(x')$ donde $\phi(x)$ es una función de los datos de entrada en un espacio vectorial de funciones que puede depender de un número finito de funciones que formen una base en dicho espacio vectorial. Esto significa que cuando usamos un Kernel implícitamente estamos haciendo un producto escalar en cierto espacio vectorial de funciones. La ventaja es que cuando utilizamos el Kernel para evaluarlo, sin necesidad de pasarlo por la función $\phi(x)$ y el producto escalar, nos da igual la dimensión que tenía el espacio vectorial. Esto significa que, si tuviéramos una base muy grande con mucha variedad de funciones, los vectores $\phi(x)$ y $\phi(x')$ además de costosos de calcular tendrían un número de elementos tan grande que sería inviable calcular su producto escalar. Sin embargo, al evaluar simplemente el Kernel obtenemos ese mismo resultado con un número de operaciones que no depende del tamaño del

espacio vectorial, por lo que podemos tomar de forma implícita bases de un tamaño extremadamente grande o incluso infinito [3, Capítulo 6].

Lo que viene a decir el párrafo anterior es que si en un modelo de regresión lineal [3, Capítulo 3] tomamos como funciones en el espacio vectorial, funciones lineales o sinusoidales, por ejemplo, solo podremos obtener como resultado de la regresión funciones lineales o sinusoidales. Es decir, el tipo de funciones resultantes viene limitado por el tipo de funciones que tomemos como base del espacio vectorial $\phi(x)$ mientras que, en el caso de usar Kernel, como ya se ha visto que pueden ser equivalentes a espacios vectoriales de funciones de tamaño infinito, no tenemos limitaciones de este tipo sobre las funciones resultantes.

Un Ejemplo Ilustrativo

Para entender mejor como funcionan los Kernel y los efectos que estos producen en las predicciones vamos a estudiar un Kernel muy conocido y utilizado, el **Square Exponential (SE)**, que es un Kernel paramétrico que viene dado por la siguiente función de covarianza.

$$\kappa_{SE}(x, x') = w^2 e^{-(x-x')^2/2l} + \sigma_y^2 \delta_{pq} \tag{2.15}$$

este kernel tiene tres parámetros: w, que controla la amplitud de la función, l, que está relacionado con la suavidad de la función, y σ_y^2 que representa el ruido de las observaciones.

Vamos a ver varios ejemplos de regresión de una función simple siguiendo la teoría explicada en 2.2.4 en la que veamos como influyen los valores de los parámetros en las predicciones realizadas sobre los datos.

En los tres ejemplos se predice el valor de la misma función de la que tenemos seis observaciones y queremos predecir su valor en todo un intervalo. El código de *Julia* que ha generado dichos ejemplos puede encontrarse en A.

En rojo está representada la media de la distribución estimada en cada punto y en verde las desviaciones típicas de la misma alrededor de la media.

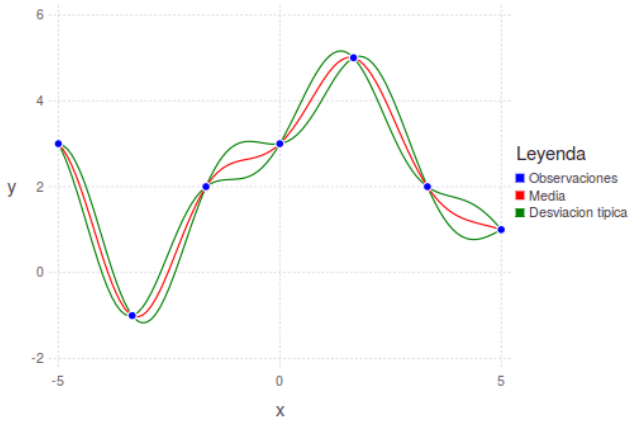

Figura 2.1: Ejemplo SE 1. $w = 1, l = 1, \sigma_y^2 = 0.00001$.

En esta primera figura se ha puesto un ruido muy reducido, por lo que las medias y las varianzas se ajustan mucho a los datos de entrenamiento cuando se acercan a ellos. Al tener los demás parámetros a 1, tanto la suavidad como la amplitud de la función se mantienen de una forma estándar.

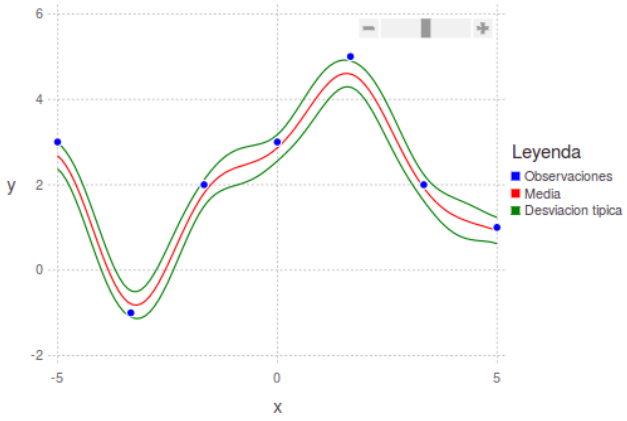

Figura 2.2: Ejemplo SE 2. $w = 0.5, l = 1, \sigma_y^2 = 0.05$.

En esta segunda imagen se ha aumentado el ruido y, como resultado, la función media ya no pasa exactamente por todas las observaciones, como era de esperar. Además se ha reducido el parámetro w que modelaba la amplitud, lo que provoca que las funciones de desviación típica representadas estén más próximas a la de la media.

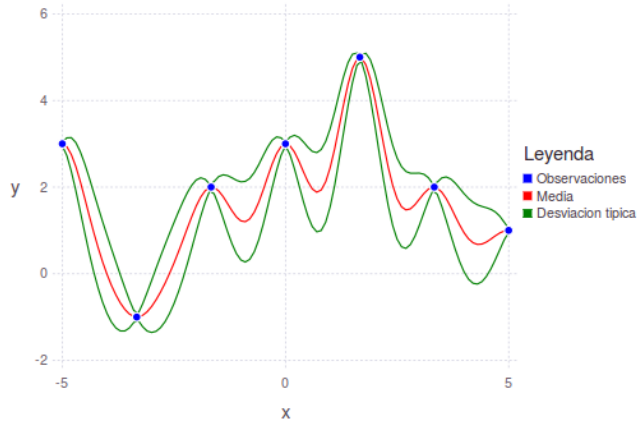

Figura 2.3: Ejemplo SE 3. $w = 1, l = 0.5, \sigma_y^2 = 0.001$.

En este tercer ejemplo lo que se ha variado es el parámetro l que era el que modelaba la suavidad de la función, dejando el ruido reducido y el parámetro w a 1 y como puede observarse, la función se vuelve mucho menos suave que en los casos anteriores.

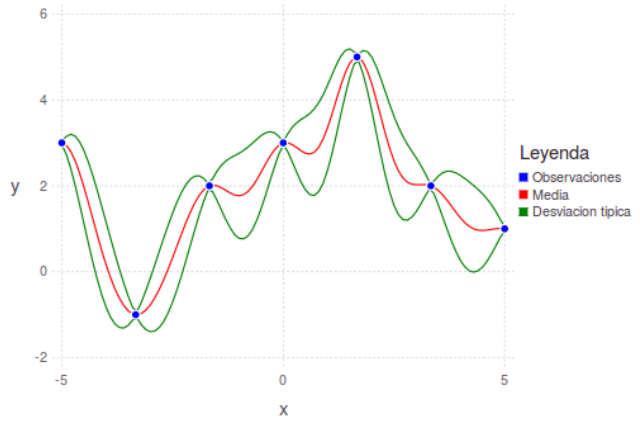

Figura 2.4: Ejemplo SE 4. $w = 1.5, l = 0.6, \sigma_y^2 = 0.001$.

En esta última imagen se ha aumentado el parámetro w que modelaba la amplitud y se ha disminuido el parámetro l que modelaba la suavidad. El resultado es que la función media obtenida es menos suave que en el caso de la figura 2.1 y las desviaciones típicas se separan más de la media que en el caso de la figura 2.3.

2.2.6 Aprendizaje de los hiperparámetros

Al igual que ocurre en el caso del SE, muchos de los Kernel utilizados van en función de ciertos parámetros de los cuales no se tiene por qué saber cuáles son los valores que se adaptan mejor a las funciones que se están intentando predecir. Vamos a ver en esta sección cómo se pueden encontrar los valores de los parámetros que mejor se adaptan a la función que se quiere predecir de forma automática.

Para entender bien cómo funciona el aprendizaje de los hiperparámetros, vamos a centrarnos en el caso del Kernel SE cuya función viene dada por 2.15. En este caso tenemos que el conjunto de nuestros hiperparámetros es

$$\theta = \{w, l, \sigma_y^2\}$$

Para poder estimar los valores de los parámetros, el modelo de los GP utiliza el criterio Maximum A Posteriori (MAP) que maximiza la verosimilitud marginal de los datos a posteriori, es decir, maximiza los valores de los parámetros que harían en nuestra función que se hayan obtenido los datos que hemos obtenido a posteriori.

Según el *Teorema de Bayes* visto en 2.6, podemos representar la probabilidad de los hiperparámetros como

$$\mathbb{P}(\theta|\boldsymbol{X}, \boldsymbol{y}) = \frac{\mathbb{P}(\boldsymbol{y}|X, \theta)\mathbb{P}(\theta)}{\mathbb{P}(\boldsymbol{y}, \boldsymbol{X})} \tag{2.16}$$

En este caso $\mathbb{P}(y|X, \theta)$ representa la verosimilitud marginal o *marginal likelihood* de los datos observados y es, por lo tanto, la función que queremos maximizar y $\mathbb{P}(\theta)$ representa la probabilidad a priori de los hiperparámetros. Normalmente se trabaja con la suposición de que todos los valores de los hiperparámetros tienen la misma probabilidad a priori, o lo que es lo mismo, siguen una distribución uniforme en todo el rango de posibles valores.

Para el cálculo de la verosimilitud marginal simplemente hay que resolver la siguiente ecuación [6, Sección 2.2.7].

$$\mathbb{P}(\boldsymbol{y}|\boldsymbol{X}, \theta) = \int \mathbb{P}(\boldsymbol{y}|f, \boldsymbol{X}, \theta)\mathbb{P}(f|\boldsymbol{X}, \theta)df \tag{2.17}$$

Como estamos en el modelo de los GP, el prior sigue una distribución normal dada por $\boldsymbol{y} \sim \mathcal{N}(0, \boldsymbol{K}_y)$, siendo $\boldsymbol{K}_y = \boldsymbol{K} + \sigma_y^2 \boldsymbol{I}_N$ en el caso de que exista ruido, por lo que podemos calcular el analíticamente el logaritmo de la ecuación dada por (2.17) como

$$log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X}, \theta)) = -\frac{1}{2}\left(\boldsymbol{y}^T \boldsymbol{K}_y^{-1} \boldsymbol{y} + log|\boldsymbol{K}_y| + Nlog(2\pi)\right) \tag{2.18}$$

donde N representaba el tamaño de la muestra observada. Utilizamos el logaritmo porque se maximiza en los mismos valores que la función original y tiene una expresión analítica más sencilla, lo que permite que puedan realizarse más rápido los cálculos.

Por lo tanto, para encontrar los parámetros que maximizan la verosimilitud de la muestra encontrada, simplemente tenemos que maximizar (2.18) (o minimizar su negativa puesto que muchos de los métodos de las librerías estándar de optimización, como es el caso de *Optim* en *Julia* [9], trabajan con métodos de descenso del gradiente).

Para hacer más eficiente y precisa la optimización, vamos a trabajar también con el gradiente de 2.18, que viene dado explícitamente por [6, Sección 5.4.1]:

$$\frac{\partial(log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X},\theta)))}{\partial\theta_j} = \frac{1}{2}\left(\boldsymbol{y}^T\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}}{\partial\theta_j}\boldsymbol{K}_y^{-1}\boldsymbol{y} - tr\left(\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}^T}{\partial\theta_j}\right)\right) \tag{2.19}$$

que es la derivada del logaritmo de la verosimilitud marginal. Es importante mencionar que $\boldsymbol{K} \neq \boldsymbol{K}_y = \boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N$ y que $\frac{\partial\boldsymbol{K}}{\partial\theta_j}$ representa la derivada de la matriz de covarianzas $\boldsymbol{K}$ respecto del parámetro θ_j, que se calcula, como veremos en ejemplos más adelante, haciendo simplemente la derivada de la función de covarianzas o Kernel para cada posición de la matriz.

2.3 Julia

Como ya se ha mencionado anteriormente, toda la implementación de los algoritmos expuestos en este trabajo se ha realizado en el lenguaje de programación **Julia** [8].

El lenguaje Julia nace en 2012 con el objetivo de posicionarse como uno de los más utilizados en los campos de computación de altas prestaciones y debe su nombre al matemático francés Gaston Julia, conocido por sus descubrimientos en el campo de los fractales.

Julia es un lenguaje de programación de alto nivel con una sintaxis bastante simple orientado a aplicaciones del campo del Cálculo Numérico y Aprendizaje Automático, aunque ha demostrado ser muy eficiente para otras tareas de programación en general.

Es un lenguaje multiparadigma de tipado dinámico, con un sofisticado compilador que permite ejecución en paralelo y distribuida y permite hacer directamente llamadas a funciones de *C* o *Fortran* [8].

Algunas de sus librerías básicas, como las de álgebra lineal, generación de números aleatorios o procesamiento de señales, llaman directamente a las librerías de C. Es por ello que obtiene unos rendimientos muy similares a los del celebérrimo lenguaje de programación, pero en el marco de una sintaxis de alto nivel equiparable a la de otros lenguajes de programación de aplicaciones similares como pueden ser *Matlab* o *R*.

En la siguiente tabla puede verse una comparación del rendimiento de algunas de las funciones más conocidas utilizadas en distintos lenguajes de programación [8]. La comparación se hace con el lenguaje C, es decir, el rendimiento de C sería 1.0 en todas las filas.

	Fortran gcc 5.1.1	**Julia** 0.4.0	**Python** 3.4.3	**R** 3.2.2	**Matlab** R2015b	**Octave** 4.0.0	**Mathematica** 10.2.0	**JavaScript** V8 3.28.71.19	**Go** go1.5	**LuaJIT** gsl-shell 2.3.1	**Java** 1.8.0_45
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

Figura 2.5: Comparación de rendimiento de distintos lenguajes con C. El rendimiento de C es 1.0.

Como puede observarse, el rendimiento de Julia esta muy próximo a C como ya mencionábamos anteriormente, superando con creces a algunos de los lenguajes más populares utilizados en el campo del Aprendizaje Automático.

3 DESARROLLO E IMPLEMENTACIÓN

En este capítulo vamos a desarrollar y explicar todas las modificaciones y adaptaciones realizadas al modelo teórico de los GPs explicado en la sección anterior para adaptarlos al problema de la **reconstrucción de imágenes**. En primer lugar explicaremos los detalles sobre el Kernel utilizado y más adelante veremos cómo hacer más eficiente los GPs aprovechando la estructura de los datos en el caso de las imágenes.

3.1 Kernels superexpresivos: *Spectral Mixture*

Como ya se ha mencionado anteriormente, la elección del Kernel de un GP es probablemente la toma de decisión más importante para construirlo, ya que estos cuantifican la similitud entre los datos de entrada del GP y definirán cómo varían las funciones de distribución estimadas con los datos de entrada.

Se consideran **kernels superexpresivos**, aquellos kernel paramétricos que tienen la capacidad de adaptarse a muchos problemas distintos, dando una buena fiabilidad en los resultados, siempre y cuando se tengan un número de datos de muestra suficientemente grande, es decir, adaptar los valores de sus parámetros para obtener predicciones con mucha eficacia en muchos problemas de regresión distintos y de una complejidad arbitraria.

3.1.1 Spectral Mixture

Los kernel que utilizaremos en este caso son denominados **Spectral Mixture (SM) kernel**, que se construyen, apoyándose en el *Teorema de Bochner* a partir de combinaciones lineales de otros kernels [1]. Esta combinación sigue teniendo todas las propiedades necesarias para seguir siendo una función de covarianzas válida y además tiene la ventaja de admitir covarianzas negativas. Mientras que en los casos de interpolación normalmente basta coger covarianzas positivas, en el caso de extrapolación es necesario poder tomar en muchas ocasiones covarianzas negativas.

Otra particularidad de este tipo de kernel es que también incorporan periodicidades en las covarianzas, lo que los hace más eficientes para aprender conjuntos de datos con ciertas regularidades. La expresión de un SE kernel es la siguiente.

$$\kappa_{SM}(x, x') = \sum_{a=1}^{A} w_a^2 e^{(-2\pi^2(x-x')^2\sigma_a^2)} cos(2\pi(x - x')\mu_a) \tag{3.1}$$

donde los parámetros w_a y σ_a aparecen elevados al cuadrado para asegurarnos que solamente tomen valores positivos, ya que si no podrían anularse unos términos con otros.

En este tipo de kernel, A determina el número de componentes del SM, es decir, el número de combinaciones de SE que se han utilizado. Con un número suficientemente alto de componentes, puede aprenderse cualquier tipo de función, sin embargo, si ponemos un número muy alto para un problema que no lo requiera, además de ser más costoso computacionalmente, podremos caer en un sobreentrenamiento del modelo.

Ejemplo de SM Kernel

Vamos a hacer un ejemplo de un GP utilizando un SM kernel y el aprendizaje de los hiperparámetros explicado en la sección 2.2.6 para entender mejor tanto el funcionamiento del SM kernel como del aprendizaje de los hiperparámetros. En este caso vamos a interpolar la función:

$$f(x) = sen(x) + 0.2cos(\frac{x}{2})$$

de la que tenemos 100 observaciones en una malla equiespaciada en el intervalo $[-8, 8]$ y queremos hallar las predicciones para los datos de test que serán 200 valores equiespaciados en el intervalo $[-20, 20]$. En este caso se ha cogido $A = 10$, es decir, 10 componentes para el SM utilizado.

Para la optimización, se ha utilizado el algoritmo *L-BFGS* por su rápida convergencia. En este caso no se ha dado el gradiente porque al tratarse de un problema simple en el que el número de datos no era excesivamente grande, se consiguen buenos resultados de forma rápida sin necesidad del gradiente ya que este último es aproximado por diferencias. Lo mismo ocurre con la inicialización de los hiperparámetros, que son los valores en los que empezará la búsqueda el algoritmo de optimización. En este caso particular, donde la función a predecir es relativamente simple y el número de datos de train bastante bajo, no es demasiado importante. Sin embargo, como veremos más adelante, en problemas más grandes la inicialización de los parámetros es muy relevante respecto a los resultados obtenidos. En este caso para la inicialización hemos utilizado para todos los parámetros una distribución $\mathcal{N}(0, 1)$.

Aunque un GP nos da tanto la media como la varianza para cada punto que se predice, en este caso tenemos que extrapolar un solo valor y por lo tanto pondremos el valor de la media, ignorando la varianza. Esto en general se hace siempre que se utilicen GPs para regresión. Los resultaos obtenidos son los siguientes.

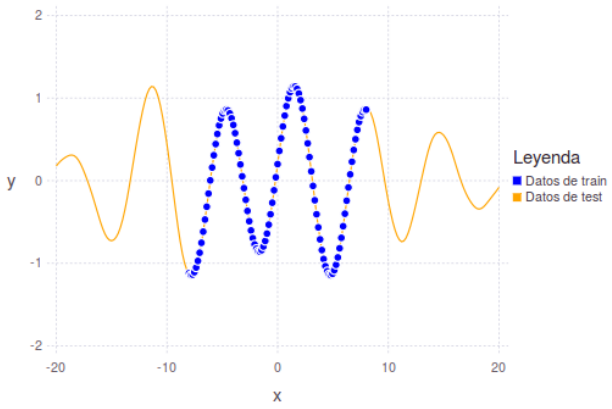

Figura 3.1: Ejemplo de regresión con SM

Los puntos azules indican los valores de muestra que se han utilizado para la calibración del modelo y la línea verde representa los valores estimados por el GP. Como puede verse, el modelo aprende de forma muy precisa la función y lo ha conseguido en un tiempo despreciable. El código utilizado para generar este ejemplo puede encontrarse en el apéndice B.

3.1.2 Spectral Mixture Multidimensional

Hasta ahora solo hemos visto Kernels unidimensionales, sin embargo, es posible construir kernels multidimensionales a partir de otros unidimensionales como ocurre en el caso del SM [10]. De nuevo, estas funciones de covarianzas multidimensionales siguen cumpliendo todas las propiedades necesarias para ser un kernel dadas por el *Teorema de Bochner* [1].

En el caso de estar trabajando en un espacio P-dimensional, el SM multidimensional (Spectral Mixture Product (SMP)) viene dado por la siguiente expresión

$$\kappa_{SMP}(\boldsymbol{\tau}|\boldsymbol{\theta}) = \prod_{p=1}^{P} \kappa_{SM}(\tau_p|\boldsymbol{\theta}_p)$$

es decir,

$$\kappa_{SMP}(\boldsymbol{\tau}|\boldsymbol{\theta}) = \prod_{p=1}^{P}\sum_{a=1}^{A} w_a^2 e^{(-2\pi^2\tau_p^2\sigma_a^2)} cos(2\pi\tau_p\mu_a) \tag{3.2}$$

donde $\boldsymbol{\tau} = x - x' \in \mathbb{R}^p$ y τ_p es la p-ésima componente de ese vector, $\boldsymbol{\theta} = \{\boldsymbol{\theta}_p\}_{p=1}^{P}$ y $\boldsymbol{\theta}_p = \{w_a, \sigma_a, \mu_a\}_{a=1}^{A}$.

En los SMP, tomaremos la suposición de que los parámetros afectan exclusivamente de una dimensión. Esta suposición, como veremos, además de ser certera, facilita enormemente los cálculos, pues cuando calculemos las derivadas respecto de los parámetros para poder usar el gradiente para mejorar la optimización, una gran cantidad de términos se anularán.

Este kernel será el que utilicemos en los GP para poder extrapolar y reconstruir las imágenes.

3.2 Aprovechando la estructura de las imágenes

El hecho de tener que optimizar la verosimilitud marginal, dada por la ecuación (2.18), implica, entre otras cosas, tener que invertir la matriz $\boldsymbol{K}_y = \boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N$, lo que conlleva, en general, un coste de $\mathcal{O}(N^3)$ y teniendo en cuenta que esta evaluación hay que hacerla muchísimas veces durante la maximización de la verosimilitud, esto hace que los GPs se conviertan en un método computacionalmente inviable para conjuntos de datos grandes, como puede ser una imagen (una imagen de 800x600 tendría un total de 480000 datos). En esta sección vamos a ver como podemos aprovechar ciertas propiedades de las imágenes para reducir más que considerablemente el coste de procesamiento del GP, llegando a costes de $\mathcal{O}(N)$ para la inversión de la matriz mencionada anteriormente.

3.2.1 Estructura Grid

La principal propiedad de las imágenes, al igual que pasa con otros problemas como todos aquellos que envuelvan localizaciones geográficas, es que sus datos vienen dados en una malla o **grid Cartesiano**, es decir, pueden representarte de la siguiente forma:

$$X = X^{(1)} \times \cdots \times X^{(D)} \tag{3.3}$$

donde $\times$ representa el producto Cartesiano y $X^{(i)}$ representa los valores de los datos para la dimensión i. En el caso de las imágenes tenemos que $D = 2$, es decir tenemos dos dimensiones y todos los los puntos de la imagen puede darse como un valor para cada una de las dimensiones.

El algoritmo que vamos a utilizar para aprovechar esta estructura grid, funciona con cualquier función de covarianza o Kernel que tenga la propiedad de ser un *kernel de producto tensorial*, es decir, que pueda representarse como un producto de otros kernels definidos positivo que toman valores de entrada en una sola dimensión. Esto significa que son aquellos kernel que se evalúen en un espacio de dimensión D y puedan darse como un producto tensorial de la siguiente forma:

$$\kappa(x_i, x_j) = \prod_{d=1}^{D} \kappa_d(x_i^{(d)}, x_j^{(d)}) \tag{3.4}$$

En los casos en los que tengamos este tipo de Kernel, la matriz de covarianzas $\boldsymbol{K}$ puede expresarse de la siguiente manera [7, Sección 5.2]:

$$\boldsymbol{K} = \bigotimes_{d=1}^{D} \boldsymbol{K}_d \tag{3.5}$$

siendo $\bigotimes$ el *producto de Kronecker* de matrices, $\boldsymbol{K}$ la matriz de covarianzas de tamaño $N \times N$ del conjunto de datos de entrenamiento que viven en un grid Cartesiano y $\boldsymbol{K}_d$ cada una de las matrices de la covarianza unidimensional dada por κ_d para cada dimensión, de tamaño $G_d \times G_d$.

El resultado 3.5, es el resultado fundamental en torno al cual gira todo nuestro algoritmo.

3.2.2 Propiedades del Producto Kronecker

Para poder aprovecharnos completamente de la estructura grid explicada en la sección anterior vamos a necesitar ciertas propiedades muy ventajosas que tiene el producto de Kronecker de matrices [7, Sección 5.2].

Teorema 3.1. *Sean $\boldsymbol{K}$ y $\boldsymbol{K}_d$ como las definidas en 3.2.1 entonces se cumple*

$$\boldsymbol{K}^{-1} = \bigotimes_{d=1}^{D} \boldsymbol{K}_d^{-1} \tag{3.6}$$

Teorema 3.2. *Sean $\boldsymbol{K}$ y $\boldsymbol{K}_d$ como las definidas en 3.2.1, sea $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$ la descomposición en autovectores y autovalores de la matriz $\boldsymbol{K}$ y sea $\boldsymbol{K}_d = \boldsymbol{Q}_d\boldsymbol{\Lambda}_d\boldsymbol{Q}_d^T$ la descomposición en autovectores y autovalores de las matrices $\boldsymbol{K}_d$, entonces se cumplen las siguientes igualdades:*

$$\boldsymbol{Q} = \bigotimes_{d=1}^{D} \boldsymbol{Q}_d \tag{3.7}$$

$$\boldsymbol{\Lambda} = \bigotimes_{d=1}^{D} \boldsymbol{\Lambda}_d \tag{3.8}$$

Teorema 3.3. *Sean $\boldsymbol{K}$ y $\boldsymbol{K}_d$ como las definidas en 3.2.1 entonces se cumplen las siguientes ecuaciones:*

$$tr(\boldsymbol{K}) = \prod_{d=1}^{D} tr(\boldsymbol{K}_d) \tag{3.9}$$

$$log(det(\boldsymbol{K})) = \sum_{d=1}^{D} G_d log(det(\boldsymbol{K}_d)) \tag{3.10}$$

$$diag(\boldsymbol{K}) = \bigotimes_{d=1}^{D} diag(\boldsymbol{K}_d) \tag{3.11}$$

3.2.3 Producto de matrices con propiedad Kronecker

Otra mejora muy importante que obtendremos en nuestro algoritmo se consigue gracias a los siguientes resultados.

Si tengo una matriz $\boldsymbol{A}$ de tamaño $N \times N$ que se puede expresar como un producto de Kronecker de matrices tal que $\boldsymbol{A} = \bigotimes_{d=1}^{D} \boldsymbol{A}_d$, y un vector $\boldsymbol{b}$ de tamaño N, existe un algoritmo que denominaremos **kron_mvproduct**, que permite evaluar el producto $\boldsymbol{A} * \boldsymbol{b}$ en un tiempo de $\mathcal{O}(N)$ en lugar del $\mathcal{O}(N^2)$ que cabría esperar de forma general [7]. El código de este algoritmo en *Julia* es el siguiente.

```
#X = [X1, X2, ... , Xn] la descomposicion en productos kronecker
function kron_mvproduct(X, b)
    n_dims = length(X)
    N = length(b)
    x = b
    for i=n_dims:-1:1
        Gd = size(X[i])[1]
        #Recolocamos el input para poder multiplicarlo por la matriz
        x_mod = reshape(x, Gd, convert(Int64, N/Gd))
        z = X[i]*x_mod
        x = reshape(z', length(z), 1)
    end
    x
end
```

Del mismo modo, si tengo una matriz $\boldsymbol{A}$ de tamaño $N \times N$ que se puede expresar como un producto de Kronecker de matrices tal que $\boldsymbol{A} = \bigotimes_{d=1}^{D} \boldsymbol{A}_d$, y una matriz $\boldsymbol{B}$ de tamaño $N \times M$, existe un algoritmo que denominaremos **kron_mmproduct** y que se basa en el algoritmo visto anteriormente *kron_mvproduct*, que permite evaluar el producto $\boldsymbol{A} * \boldsymbol{B}$ en un tiempo de $\mathcal{O}(NM)$ en lugar del $\mathcal{O}(N^2M)$ que cabría esperar de forma general y en un espacio de $\mathcal{O}(NM)$. El código de este algoritmo en *Julia* es el siguiente.

```
#X = [X1, X2, ... , Xn] la descomposicion en productos kronecker
function kron_mmproduct(X, A)
    N, M = size(A)
    res = kron_mvproduct(X, hcat(A[:,1]))
    for i=2:M
        res=hcat(res, kron_mvproduct(X, hcat(A[:,i])))
    end
    res
end
```

3.2.4 Mejorando el algoritmo de los GPs

En esta sección vamos a ver cómo utilizar todas las propiedades vistas en los apartados anteriores para mejorar el algoritmo de los GP cuando tenemos los datos con una estructura grid. Para poder predecir utilizando GP, tenemos que poder evaluar la verosimilitud marginal y su derivada, que venían dadas por las ecuaciones (2.18) y (2.19) respectivamente.

Evaluación la Verosimilitud Marginal

En el caso de la verosimilitud marginal tenemos los siguientes términos.

$$log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X},\theta)) = -\frac{1}{2}\left(\underbrace{\boldsymbol{y}^T\boldsymbol{K}_y^{-1}\boldsymbol{y}}_{A} + \underbrace{log|\boldsymbol{K}_y|}_{B} + \underbrace{Nlog(2\pi)}_{C}\right)$$

A: Recordemos que $\boldsymbol{K}_y = \boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N$, luego para evaluar este término, utilizaremos la siguiente identidad de matrices que proviene de la descomposición en autovalores y autovectores de una matriz. Sea $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$, entonces se cumple:

$$\boldsymbol{\alpha} := (\boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N)^{-1}\boldsymbol{y} = \boldsymbol{Q}(\Lambda + \sigma_y^2\boldsymbol{I}_N)^{-1}\boldsymbol{Q}^T\boldsymbol{y} \tag{3.12}$$

que podemos expresarlo, utilizando (3.7) y (3.8) como

$$\boldsymbol{\alpha} = \bigotimes_{d=1}^{D}(\boldsymbol{Q}_d)(\bigotimes_{d=1}^{D}(\boldsymbol{\Lambda}_d) + \sigma_y^2\boldsymbol{I}_N)^{-1}\bigotimes_{d=1}^{D}(\boldsymbol{Q}_d^T)\boldsymbol{y} \tag{3.13}$$

donde todas las multiplicaciones de matrices por vector pueden realizarse utilizando la rutina *kron_mvproduct* definida en 3.2.3 en un tiempo y espacio de $\mathcal{O}(N)$.

B: En este caso queremos evaluar $log|\boldsymbol{K}_y| = log|\boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N|$, para ello utilizamos la propiedad dada por (3.10) y propiedades básicas de las matrices [5] y tendremos la siguiente igualdad:

$$log|\boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N| = \sum_{\lambda\in\Lambda}(\lambda + \sigma_y^2) \tag{3.14}$$

que de nuevo puede hacerse en un tiempo y espacio de $\mathcal{O}(N)$.

C: Es simplemente un producto donde N representa el tamaño de la muestra de los datos de train y cuyo tiempo de ejecución puede considerarse despreciable.

El código en *Julia* de la evaluación de la verosimilitud marginal es el siguiente:

```
function marginal_likelihood(X, y, phi, A)
    n_dims = length(X)
    n_params = length(phi)
    K_d = []
    eig_value_d = []
    eig_vector_d = []
    eig_vector_d_t = []
    eig_values = 1
    for i=1:n_dims
        curr_K = covfunc(X[i], phi[(i-1)*div(n_params,2)+1: i*div(n_params
            ,2)])
        curr_eig_value, curr_eig_vector = eig(curr_K)
        push!(K_d, curr_K)
        push!(eig_value_d, curr_eig_value)
        push!(eig_vector_d, curr_eig_vector)
        push!(eig_vector_d_t, curr_eig_vector')
        eig_values = kron(eig_values, curr_eig_value)
    end
    alpha = kron_mvproduct(eig_vector_d_t, y)
    alpha = (1./(eig_values.+sigma_n_cuadrado)).*alpha
    alpha = kron_mvproduct(eig_vector_d, alpha)
```

```
    N = length(eig_values)
    sum = mapreduce((x)->log(x+sigma_n_cuadrado), +, eig_values)
    log_Z = (-1/2*(y'*alpha + sum + N*log(2*pi)))[1]
end
```

Evaluación de la derivada de la Verosimilitud Marginal

En el caso de la derivada de la verosimilitud marginal, teníamos la siguiente expresión:

$$\frac{\partial(log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X},\theta)))}{\partial\theta_j} = \frac{1}{2}\left(\underbrace{\boldsymbol{y}^T\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}}{\partial\theta_j}\boldsymbol{K}_y^{-1}\boldsymbol{y}}_{A} - \underbrace{tr\left(\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}^T}{\partial\theta_j}\right)}_{B}\right)$$

A: Puesto que $\boldsymbol{K}_y$ es una matriz simétrica y semidifinida positiva, entonces se cumple $\boldsymbol{y}^T\boldsymbol{K}_y^{-1} = \boldsymbol{K}_y^{-1}\boldsymbol{y} = \boldsymbol{\alpha}$, por lo tanto ambos términos se calculan como en el apartado anterior.

Para el cálculo de $\frac{\partial\boldsymbol{K}}{\partial\theta_j}$ utilizaremos la siguiente igualdad [7]:

$$\frac{\partial\boldsymbol{K}}{\partial\theta_j} = \sum_{d=1}^{D}\frac{\partial\boldsymbol{K}_d}{\partial\theta_j}\otimes\left(\bigotimes_{i\neq d}\boldsymbol{K}_i\right) \tag{3.15}$$

Y teniendo en cuenta que como ya mencionamos anteriormente, los parámetros solamente dependen de una dimensión, la mayoría de las derivadas cruzadas se anulan, lo que nos permite escribir la expresión anterior como

$$\frac{\partial\boldsymbol{K}}{\partial\theta_j} = \frac{\partial\boldsymbol{K}_j}{\partial\theta_j}\otimes\left(\bigotimes_{i\neq d}\boldsymbol{K}_i\right) \tag{3.16}$$

Con esto hemos conseguido escribir $\frac{\partial\boldsymbol{K}}{\partial\theta_j}$ como un producto Kronecker de matrices, y puesto que está multiplicando a un vector ($\boldsymbol{\alpha}$), de nuevo podemos reutilizar la rutina *kron_mvproduct* definida en 3.2.3 para resolver A en un tiempo y espacio de $\mathcal{O}(N)$.

B: Para el cálculo de este término, utilizaremos las propiedades de la traza de una matriz [5] que vienen desarrolladas en la ecuación (5.35) de [7], obteniendo el siguiente resultado:

$$tr\left((\boldsymbol{K}+\sigma_y^2\boldsymbol{I}_N)^{-1}\frac{\partial\boldsymbol{K}^T}{\partial\theta_j}\right) = diag\left((\boldsymbol{\Lambda}+\sigma_y^2\boldsymbol{I}_N)^{-1}\right)^T diag\left(\boldsymbol{Q}^T\frac{\partial\boldsymbol{K}^T}{\partial\theta_j}\boldsymbol{Q}\right) \tag{3.17}$$

donde de nuevo $\boldsymbol{Q}$ lo podemos expresar en su descomposición Kronecker como ya hemos visto varias veces y podemos utilizar la propiedad 3.11 para calcular de forma más eficiente la parte de la derecha de (3.17).

Por lo tanto de nuevo podemos resolver todo el término B en un tiempo y espacio de $\mathcal{O}(N)$.

Esta expresión la tenemos que calcular para cada uno de los hiperparámetros ϕ_j, luego lo que necesitamos para poder utilizar los algoritmos de optimización de forma más eficiente es el gradiente. El código en *Julia* de la evaluación del gradiente de la verosimilitud marginal es el siguiente:

```
function d_marginal_likelihood(X, y, phi, A)
    n_dims = length(X)
    n_params = length(phi)
    K_d = []
    eig_value_d = []
    eig_vector_d = []
    eig_vector_d_t = []
    eig_values = 1
    for i=1:n_dims
        curr_K = covfunc(X[i], phi[(i-1)*div(n_params,2)+1: i*div(n_params
            ,2)])
        curr_eig_value, curr_eig_vector = eig(curr_K)
        push!(K_d, curr_K)
        push!(eig_value_d, curr_eig_value)
        push!(eig_vector_d, curr_eig_vector)
        push!(eig_vector_d_t, curr_eig_vector')
        eig_values = kron(eig_values, curr_eig_value)
    end
    alpha = kron_mvproduct(eig_vector_d_t, y)
    alpha = (1./(eig_values.+sigma_n_cuadrado)).*alpha
    alpha = kron_mvproduct(eig_vector_d, alpha)
    dlogZ_dphi = []
    for i=1:n_params
        dKd_dphi = []
        sigma_d = []
        for d=1:n_dims
            if (d==1)
                if (i>div(n_params,2))
                    dKd_dphii = zeros(length(X[d]), length(X[d]))
                else
                    dKd_dphii = dcovfunc_i(X[d], sigma_f_cuadrado, phi[(d-1)
                        *div(n_params,2)+1: d*div(n_params,2)], i)
                end
            elseif (d==2)
                if (i<=div(n_params,2))
                    dKd_dphii = zeros(length(X[d]), length(X[d]))
                else
                    dKd_dphii = dcovfunc_i(X[d], sigma_f_cuadrado, phi[(d-1)
                        *div(n_params,2)+1: d*div(n_params,2)], i-div(
                        n_params,2))
                end
            end
            sigma_curr = diag(eig_vector_d_t[d]*dKd_dphii'*eig_vector_d[d])
            push!(dKd_dphi, dKd_dphii)
            push!(sigma_d, sigma_curr)
        end
        tmparray = []
        if i>div(n_params,2)
            push!(tmparray, diag(eig_vector_d_t[1]*K_d[1]*eig_vector_d[1]))
            push!(tmparray, sigma_d[2])
        else
            push!(tmparray, sigma_d[1])
            push!(tmparray, diag(eig_vector_d_t[2]*K_d[2]*eig_vector_d[2]))
        end
        sigma = reduce(kron, tmparray)
        tmparray = []
        if i>div(n_params,2)
            push!(tmparray, K_d[1])
```

```
55              push!(tmparray, dKd_dphi[2])
56          else
57              push!(tmparray, dKd_dphi[1])
58              push!(tmparray, K_d[2])
59          end
60          K_tilde = kron_mvproduct(tmparray, alpha)
61          dlogZ_dphii = 1/2*(alpha'*K_tilde)-1/2*dot((1./(eig_values+
                sigma_n_cuadrado)),sigma)
62          push!(dlogZ_dphi, dlogZ_dphii[1])
63      end
64      dlogZ_dphi
65  end
```

Evaluación del resto de operaciones del GP

Hasta ahora hemos visto cómo calcular la verosimilitud marginal y la derivada de la verosimilitud marginal en un tiempo de $\mathcal{O}(N)$, sin embargo, este es el "cuello de botella" del algoritmo. El resto de cuentas, además de ser más simples, solo se ejecutarán una vez. De nuevo también se ejecuta en un tiempo de $\mathcal{O}(N)$.

Teniendo en cuenta que los valores predichos serán la media de la distribución en cada punto, no hace falta el cálculo explícito de la varianza de las distribuciones dadas por el GP, por lo tanto el código en *Julia* para evaluar el resto del algoritmo del GP es el siguiente.

```
1   function GP(X, Xstar, phi_opt)
2       n_dims = length(X)
3       n_params = length(phi_opt)
4       K_d = []
5       eig_value_d = []
6       eig_vector_d = []
7       eig_vector_d_t = []
8       eig_values = 1
9       for i=1:n_dims
10          curr_K = covfunc(X[i], phi_opt[(i-1)*div(n_params,2)+1: i*div(
                n_params,2)])
11          curr_eig_value, curr_eig_vector = eig(curr_K)
12          push!(K_d, curr_K)
13          push!(eig_value_d, curr_eig_value)
14          push!(eig_vector_d, curr_eig_vector)
15          push!(eig_vector_d_t, curr_eig_vector')
16          eig_values = kron(eig_values, curr_eig_value)
17      end
18      alpha = kron_mvproduct(eig_vector_d_t, y)
19      alpha = (1./(eig_values.+sigma_n_cuadrado)).*alpha
20      alpha = kron_mvproduct(eig_vector_d, alpha)
21
22      KMN_d = []
23      for i=1:n_dims
24          curr_KMN = covfunc_star(X[i], Xstar[i], phi_opt[(i-1)*div(n_params
                ,2)+1: i*div(n_params,2)])
25          push!(KMN_d, curr_KMN)
26      end
27      K_MN = kron(KMN_d[1], KMN_d[2])
28      mu_star = K_MN'*alpha
29  end
```

En el capítulo 4 podemos encontrar ejemplos de imágenes reconstruidas con el algoritmo descrito en esta sección.

3.2.5 Mejora obtenida

Como ya se ha ido mencionando a lo largo de todos los apartados de la sección anterior, en ninguno de los cálculos nos excedemos de un tiempo de $\mathcal{O}(N)$, mientras que si intentáramos extrapolar imágenes con un GP normal, sólo con el cálculo de la inversa, ya tendríamos un tiempo de $\mathcal{O}(N^3)$, por lo que la mejora, tanto en eficiencia temporal como en espacio, es extremadamente grande.

3.3 Completado de imágenes

Hasta ahora, sólo hemos estudiado cómo tratar imágenes que formen un grid completo, sin embargo, cuando queramos reconstruir una imagen, es posible que la zona a reconstruir esté por el medio, haciendo que ni los datos de train, ni los datos de test formen, en general, un grid completo. En esta sección vamos a ver cómo podemos aprovechar la estructura "parcial" de grid de dichas imágenes para poder seguir usando nuestro algoritmo de GP acelerado visto en la sección anterior.

Supongamos que estamos en el caso en el que de los N datos que formarían el grid completo, tenemos M observaciones, con valor $\boldsymbol{y}_M$. En este caso completaremos el grid con W observaciones imaginarias con la siguiente distribución $\boldsymbol{y}_W \sim \mathcal{N}(f_w, \epsilon^{-1}\boldsymbol{I}_W)$ donde $\epsilon \to 0$ y por lo tanto el ruido de los datos imaginarios añadidos será muy grande, lo que hará que no afecte negativamente a la inferencia.

Una vez añadidos las observaciones imaginarias, pasaremos a tener un vector de observaciones $\boldsymbol{y} = [\boldsymbol{y}_M, \boldsymbol{y}_W]$ de tamaño $N = M + W$ que en este caso sigue una distribución $\boldsymbol{y}_W \sim \mathcal{N}(f, D_N)$ donde la matriz de ruidos D_N viene dada por

$$D_N = \begin{bmatrix} \sigma_y^2 \boldsymbol{I}_M & 0 \\ 0 & \epsilon^{-1}\boldsymbol{I}_W \end{bmatrix} \tag{3.18}$$

donde σ_y^2 es el ruido de las observaciones reales al igual que en los apartados anteriores. Es decir, tendremos una nueva matriz de ruidos donde a cada posición le corresponde el ruido normal σ_y^2, si es una observación real la de esa posición, o ϵ^{-1}, si es una observación imaginaria.

En [11] hay una demostración de que las observaciones imaginarias añadidas no afectan a la inferencia, es decir, se cumple:

$$\lim_{\epsilon \to 0} (K_N + D_N)^{-1}\boldsymbol{y} = (K_M + D_M)^{-1}\boldsymbol{y}_M \tag{3.19}$$

En este marco, vamos a ver como tenemos que proceder para evaluar todas las operaciones del GP en este caso.

3.3.1 Adaptando el algoritmo

Gradientes Conjugados Precondicionados (PCG)

Uno de los términos que tenemos que evaluar más veces a lo largo del algoritmo del GP es $\boldsymbol{\alpha} = (\boldsymbol{K}_N + D_N)^{-1}\boldsymbol{y}$, sin embargo, no podemos ayudarnos, como hacíamos antes de la descomposición en autovectores y autovalores y utilizar las ventajas de las propiedades Kronecker porque D_N, en esta ocasión, no tiene todos los términos iguales. Por lo tanto para resolver este término utilizaremos una pequeña variante del método **Gradientes Conjugados (CG)** [2].

Lo que hace el método CG es resolver de forma iterativa un sistema lineal de la forma $A * x = b$, siendo A una matriz y b un vector columna. En este caso lo que tenemos que resolver es $\boldsymbol{z} = (\boldsymbol{K}_N + D_N)^{-1}\boldsymbol{y}$, que es equivalente al sistema $(\boldsymbol{K}_N + D_N)\boldsymbol{z} = \boldsymbol{y}$ que es el que resolveremos por nuestro método CG mejorado.

La principal ventaja es que con esto hemos podido calcular la z deseada sin necesidad de calcular la inversa (que es muy costoso), simplemente utilizando productos de matrices.

La variación del método CG consiste en lo siguiente [11]. Si utilizo la matriz $C := D_N^{-1/2}$, para precondicionar el sistema y utilizar un método de Gradientes Conjugados Precondicionados (PCG), lo que trato de resolver en este caso es el sistema

$$C(\boldsymbol{K}_N + D_N)C\boldsymbol{z} = C\boldsymbol{y} \tag{3.20}$$

que si desarrollamos podemos escribir como

$$(C\boldsymbol{K}_N C + CD_N C)\boldsymbol{z} = C\boldsymbol{y} \equiv (C\boldsymbol{K}_N C + I)\boldsymbol{z} = C\boldsymbol{y} \equiv C\boldsymbol{K}_N(C\boldsymbol{z}) + \boldsymbol{z} = C\boldsymbol{y}$$

y como C es una matriz diagonal (en nuestro caso lo tomaremos como un vector directamente), el producto $C\boldsymbol{z}$ se resuelve como un producto elemento a elemento y tendremos que de nuevo podemos aprovecharnos de la estructura Kronecker para resolver $\boldsymbol{K}_N(C\boldsymbol{z})$ y esa será precisamente la variación que introduzcamos en el algoritmo CG y que nos permitirá resolver el sistema en un tiempo mucho menor al que lo haría el método normal.

El código en *Julia* del algoritmo descrito en esta sección es el siguiente.

```
#C es una matriz diagonal pero lo paso como vector
#A es una descomposicion Kronecker
function GP_CG(A, b, tol, maxit, C)
    x0 = zeros(length(b))
    x = x0
    r = b-prod_GP_CG(C, A,x)
    p_a = []
    alfa_a = []
    x_a = []
    r_a = []
    beta_a = []
    push!(p_a, r)
    push!(p_a, r)
    push!(r_a, r)
    push!(x_a, x0)
    push!(alfa_a, 0)
    push!(beta_a, 0)
        for i=2:maxit
        alfa_i = sum(r_a[i-1].*r_a[i-1])/sum(p_a[i].*prod_GP_CG(C, A,p_a[i
            ]))
        x_i = x_a[i-1]+alfa_i*p_a[i]
        r_i = r_a[i-1]-alfa_i*prod_GP_CG(C, A,p_a[i])
        beta_i = sum(r_i.*r_i)/sum(r_a[i-1].*r_a[i-1])
        p_imas = r_i+beta_i*p_a[i]
        push!(alfa_a, alfa_i)
        push!(beta_a, beta_i)
        push!(x_a, x_i)
        push!(r_a, r_i)
        push!(p_a, p_imas)
        if all(r_i .< (tol*ones(length(b))))
```

```
            return x_i
        end
    end
    x_a[maxit]
end

#A dada en su descomposicion kronekcer A = [A1, A2 ,...], C es el vector de la diagonal
function prod_GP_CG(C, A, b)
    C .* kron_mvproduct(A, C.*b) + b
end
```

El resultado que obtenemos con el algoritmo es $z' = C^{-1}(\boldsymbol{K}_N + D_N)^{-1}\boldsymbol{y}$ por lo que para obtener el z buscado anteriormente tendremos que multiplicar el resultado del algoritmo por C.

Evaluación de la Verosimilitud Marginal

Recordemos que la función que tenemos que evaluar es la siguiente.

$$log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X},\theta)) = -\frac{1}{2}\left(\underbrace{\boldsymbol{y}^T\boldsymbol{K}_y^{-1}\boldsymbol{y}}_{A} + \underbrace{log|\boldsymbol{K}_y|}_{B} + \underbrace{Nlog(2\pi)}_{C}\right)$$

A: En este caso para evaluar este término simplemente utilizaremos el algoritmo PCG explicado en la sección anterior 3.3.1.

B: Recordemos, que en el caso de tener el grid completo este término lo resolvíamos como

$$log|\boldsymbol{K} + \sigma_y^2\boldsymbol{I}_N| = \sum_{\lambda\in\Lambda}(\lambda + \sigma_y^2)$$

En este caso, para resolverlo utilizaremos la siguiente aproximación [11]:

$$log|\boldsymbol{K}_M + D_N| = \sum_{i=1}^{M} log(\lambda_i^M + \sigma_y^2) \approx \sum_{i=1}^{M} log(\tilde{\lambda}_i^M + \sigma_y^2) \tag{3.21}$$

donde hemos aproximado los autovalores λ_i^M de la matriz $\boldsymbol{K}_M$ por los autovalores de la matriz $\boldsymbol{K}_N$ tal que $\tilde{\lambda}_i^M = \frac{M}{N}\lambda_i^N, i = 1, 2, ..., M$ que es una aproximación especialmente buena para matrices grandes ($M > 1000$).

C: De nuevo es una multiplicación con coste despreciable.

Evaluación de la derivada de la Verosimilitud Marginal

En este caso al función que tenemos que evaluar es:

$$\frac{\partial(log(\mathbb{P}(\boldsymbol{y}|\boldsymbol{X},\theta)))}{\partial\theta_j} = \frac{1}{2}\left(\underbrace{\boldsymbol{y}^T\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}}{\partial\theta_j}\boldsymbol{K}_y^{-1}\boldsymbol{y}}_{A} - \underbrace{tr\left(\boldsymbol{K}_y^{-1}\frac{\partial\boldsymbol{K}^T}{\partial\theta_j}\right)}_{B}\right)$$

A: Este término se puede evaluar de la misma forma que en el caso de que el grid estuviera completo visto en 3.2.4 con la salvedad de que para calcular $\boldsymbol{K}_y^{-1}\boldsymbol{y}$ utilizaremos el PCG visto en 3.3.1.

B: Para calcular este término vamos a utilizar la derivada de la aproximación vista en la sección anterior que era

$$\sum_{i=1}^{M} log(\lambda_i^M + \sigma_y^2) \approx \sum_{i=1}^{M} log(\tilde{\lambda}_i^M + \sigma_y^2)$$

Sea A una matriz real y simétrica y sean $\lambda_i, \boldsymbol{v}_i$ un autovalor y su autovector correspondiente, entonces la derivada del autovalor viene dada por la siguiente expresión [5]:

$$\partial \lambda_i = \boldsymbol{v}_i^T \partial(A) \boldsymbol{v}_i \tag{3.22}$$

Utilizando esta derivada de los autovalores, podemos derivar la aproximación anterior de la siguiente manera:

$$\frac{\partial \left(\sum_{i=1}^{M} log(\tilde{\lambda}_i^M + \sigma_y^2) \right)}{\partial \theta_j} = \sum_{i=1}^{M} \frac{\frac{\partial \tilde{\lambda}_i^M}{\partial \theta_j}}{\tilde{\lambda}_i^M + \sigma_y^2} = \sum_{i=1}^{M} \frac{\frac{M}{N} \boldsymbol{v}_i^T \frac{\partial \boldsymbol{K}}{\partial \theta_j} \boldsymbol{v}_i}{\frac{M}{N} \lambda_i^N + \sigma_y^2} \tag{3.23}$$

Y con esto ya tenemos evaluados todos los términos de la derivada de la verosimilitud marginal.

Evaluación del resto de operaciones del GP

El resto de las operaciones del GP se pueden evaluar exactamente de la misma forma que lo hacíamos en el caso de que el grid estuviera completo con la salvedad de que hay que usar el algoritmo PCG cuando sea necesario.

Con esto ya tenemos todos los cálculos teóricos hechos para poder hacer la extrapolación de imágenes en el caso general. En el próximo capítulo veremos algunos ejemplos de extrapolaciones producidos con el código en *Julia*.

4 VALIDACIÓN EXPERIMENTAL

En este capítulo mostraremos casos particulares de uso de los algoritmos descritos a lo largo del trabajo así como las consideraciones tomadas para la ejecución de lo mismo y los motivos que nos llevaron a tomarlas.

4.1 Imágenes completas

En la primera sección veremos ejemplos de imágenes que formaban un grid completo. Se corresponde con el algoritmo descrito en 3.2.

Como **kernel** se ha utilizado el **SMP** descrito anteriormente con un número de componentes $A = 50$, aunque probablemente se podrían haber conseguido resultados muy similares con un número menor de componentes en un tiempo menor, puesto que las imágenes no son excesivamente complejas, elegí ese número para asegurar que se pueda aprender bien la estructura de las imágenes, pues haciendo pruebas he llegado a la conclusión de que imágenes de una cierta complejidad o resolución requieren de un número mayor de componentes para poder hacer una correcta extrapolación, pero el tiempo de ejecución se vuelve bastante elevado.

Para la optimización de la **verosimilitud marginal** se ha utilizado, al igual que se ha hecho en los ejemplos anteriores, el algoritmo **L-BFGS** por su buen rendimiento cuando le pasamos el gradiente de la función a optimizar como es el caso.

La inicialización de los parámetros para empezar a maximizar la verosimilitud marginal ha demostrado ser muy influyente en el resultado final de la extrapolación. La inicialización que hemos llevado a cabo en este caso y que ha dado bastantes buenos resultado para los parámetros del SMP ha sido la siguiente [11].

- **w:** Para la inicialización de las w hemos utilizado en todas las componentes la desviación típica de los datos divido entre el número de componentes.
- σ**:** Las σ se han inicializado como la inversa de una distribución normal de media el número de elementos de la dimensión a la que corresponde la componente multiplicado por una constante.
- μ**:** La inicialización de las μ se ha llevado a cabo siguiendo una distribución uniforme entre 0 y un medio de la *Frecuencia de Nyquist*, que en este caso, al tratarse

de imágenes, no es más que la distancia cuantificada entre dos elementos contiguos, es decir 1 Pixel.

Para la inicialización del ruido hemos seguido el siguiente criterio.

$$\sigma_y^2 = (0.1(mean(abs(y))))^2$$

Así mismo, es importante que todos los datos que se pasen hayan sido normalizados, pues si no la inicialización de los parámetros del modelo no tendría demasiado sentido y además facilita el aprendizaje de la estructura de la imagen por parte del GP.

Para simplificar los cálculos debido a que el tiempo de ejecución del GP entero es bastante elevado, se han realizado las pruebas con imágenes en blanco y negro, pero sería posible reconstruir las imágenes a color si se usara un GP para cada unos de los tres colores (RGB). La reconstrucción de las imágenes de ejemplo es la siguiente.

Figura 4.1: Imagen 1 completa

Figura 4.2: Imagen 1 train

Figura 4.3: Imagen 1 test

En la 4.1 puede verse la imagen completa antes de la extrapolación. En la 4.2 puede verse la misma imagen separada en los datos de train y los datos de test hy por último en la imagen 4.3 se puede ver la imagen reconstruida por el GP, como puede verse la diferencia con la imagen original es prácticamente nula.

Un segundo ejemplo de extrapolación en el caso de imágenes que tengan una estructura grid completa es el siguiente.

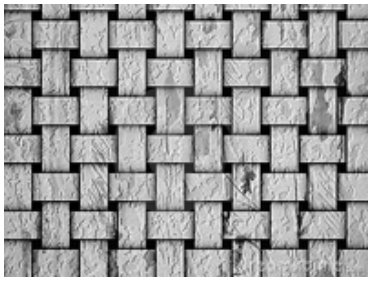

Figura 4.4: Imagen 2 completa

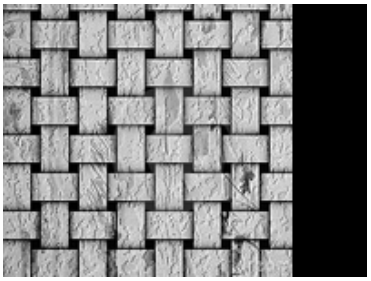

Figura 4.5: Imagen 2 train

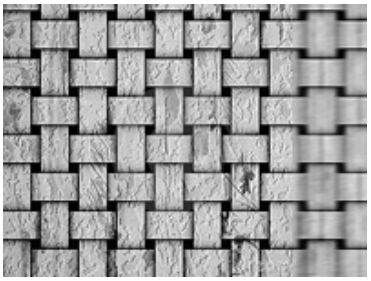

Figura 4.6: Imagen 2 test

En este caso la reconstrucción no es tan precisa debida a que la imagen tiene una complejidad mayor, pero podría haberse conseguido una reconstrucción casi perfecta con un número mayor de componentes en el kernel y el correspondiente coste computacional.

4.2 Imágenes incompletas

En esta sección mostraremos algunos ejemplos de imágenes reconstruidas utilizando el algoritmo descrito en la sección 3.3.

Debido a la necesidad de uso del algoritmo PCG para aproximar el resultado de $(\boldsymbol{K}_N + D_N)^{-1}(\boldsymbol{y})$ el coste computacional de este algoritmo es mucho mayor, por lo que

solo exhibiremos dos ejemplos simples de imágenes con bastante regularidad, ya que la extrapolación de imágenes complejas y en buena definición sería muy costosa, aunque con un número de componentes adecuado y una capacidad de computo suficiente se podría extrapolar, con una precisión muy grande, prácticamente cualquier imagen.

Todas las inicializaciones y consideraciones del algoritmo son exactamente las mismas que las utilizadas en la sección anterior a las que además añadiremos el ruido de las observaciones imaginarias que tendrá un valor de $\epsilon^{-1} = 10^5$.

En primer lugar vemos un ejemplo de un patrón de cuadros blancos y negros, cuya estructura es fácilmente extrapolable, por lo que en el número de componentes del SMP hemos elegido simplemente $A = 10$, que es un número más que suficiente para este ejemplo.

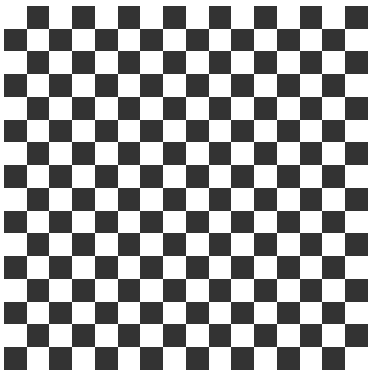

Figura 4.7: Imagen 3 completa

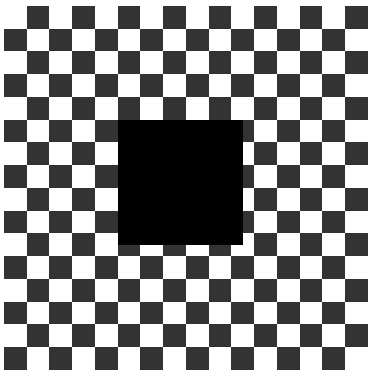

Figura 4.8: Imagen 3 train

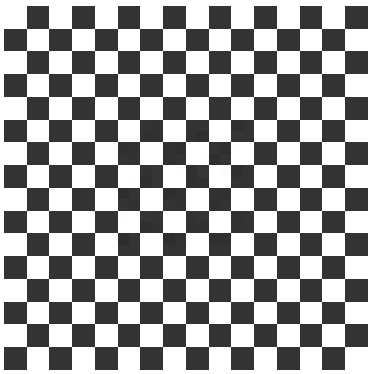

Figura 4.9: Imagen 3 test

Por último exhibiremos un ejemplo con un muro de ladrillos que tiene un patrón bastante regular y fácilmente extrapolable pero donde puede verse que los resultados son prácticamente perfectos en comparación con la imagen original. De nuevo se ha utilizado la imagen en blanco y negro pero podría reconstruirse a color utilizando tres GP, uno para cada color, en lugar de uno solo.

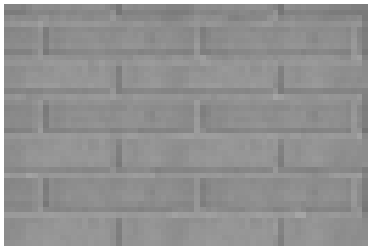

Figura 4.10: Imagen 4 completa

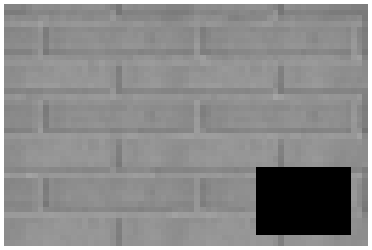

Figura 4.11: Imagen 4 train

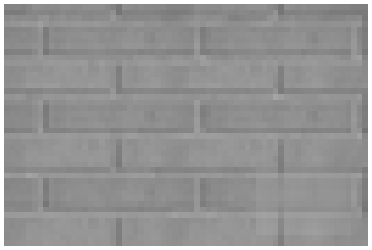

Figura 4.12: Imagen 4 test

5 Conclusiones

Con este trabajo se ha propuesto una gran aproximación al problema de extrapolación de las imágenes. En vista de los resultados obtenidos pueden extraerse importantes conclusiones acerca de los GP, el problema de la extrapolación de las imágenes y el *Aprendizaje Supervisado* en general.

En primer lugar, queda patente el potencial de los GP para la regresión en casos donde la cantidad de datos a analizar no es tan pequeño a través de la explotación de la estructura subyacente en el conjuntos de datos. Aunque en este caso particular solo lo hayamos hecho con estructuras grid y en el caso de las imágenes, el resultado aquí mostrado abre las puertas al estudio de otras muchas estructuras en distintos conjuntos de datos con el fin de poder mejorar el rendimiento, tanto de los GP como de otros algoritmos de Aprendizaje Automático.

En segundo lugar, el hecho de haber podido completar la estructura grid de las imágenes en los casos en los que esta no era completa, es una clara demostración de que muchos conjuntos de datos pueden tener una organización parcial, aunque a primera vista no se vea, y que puede completarse por medio de diferentes técnicas para poder tener datos totalmente estructurados. Esto puede servir como ejemplo para muchos problemas del Aprendizaje Automático en los que haciendo un buen estudio de los datos a utilizar, quizá puedan lograrse grandes avances y mejoras.

Dentro del mundo de los Procesos Gaussianos, los cuales han demostrado tener un potencial extremadamente grande para problemas de regresión, hemos podido comprobar la importancia de la elección del Kernel y sus parámetros y cómo varían las predicciones realizadas con las variaciones en los kernel.

Además, hemos podido observar que tienen un especial potencial los Kernel Superexpresivos, pues como hemos visto, para un número de componentes suficientemente grande (con la capacidad de computo correspondiente), puede aprenderse la estructura de imágenes muy complejas y grandes. Esto abre las puertas a la investigación de resolución de otros muchos problemas del Aprendizaje Supervisado mediante GPs que utilicen Kernel Superexpresivos, pues es cierto que muchos de los problemas más frecuentes del Aprendizaje Automático no tienen una complejidad tan grande como tienen las imágenes y en este caso hemos podido aprenderlas.

Otra conclusión que puede extraerse del trabajo, es el hecho de que hemos utilizado un algoritmo que ya existía para resolver un problema que a priori no podría solucionar

por sus altos costes, como es el caso de extrapolación de imágenes que no tuvieran una estructura de rejilla completa. De aquí puede concluirse que no siempre es necesario reinventar la rueda para resolver un problema, en ocasiones basta con dar una vuelta de tuerca a las herramientas ya disponibles.

Por último, y probablemente más importante, ha quedado demostrado, una vez más, la importancia de hacer un análisis previo de los datos antes de utilizar algún algoritmo de Aprendizaje Automático, pues estos nos pueden llevar a hacer predicciones con una precisión mucho mayor o en un tiempo infinitamente más pequeño.

GLOSARIO

datos de test En el campo de aprendizaje automático, los datos de test son los utilizados para comprobar la eficacia del modelo una vez haya sido creado y calibrado.. 9

delta de kronecker Función que toma valor 1 cuando sus dos argumentos son iguales y 0 en caso contrario.. 8

grid Una estructura grid es una estructura en la que los datos vienen representados en una malla equiespaciada, como ocurre en el caso de las imágenes que se pueden representar como una matriz de pixels.. 17, 18, 19

Kernel Es una función de covarianza entre dos valores de entrada que dice que grado de relación en función de ciertos criterios intrínsecos al propio kernel tienen dichos valores.. 1, 2, 8, 9, 10, 12, 13, 15, 17, 18

movimientos rigidos Se conocen como movimientos rigidos todas aquellas que mantienen las distancias.. 9

Pixel Unidad mínima de división de una imagen. 2, 29

posterior Función final construida a partir del prior y de los datos observados.. 8

prior Es la función de la cual parte el GP para construir una función final o posterior a partir de esta y los datos observados. El posterior heredará propiedades de esta función.. 8, 12

proceso estocástico Sucesión de variables aleatorias que evolucionan en función de otra variable y que tiene cada una su propia función de distribución de probabilidad.. 1

regresión Es el proceso por el cual se intenta establecer una relación de dependencia entre una o más variables independientes y una o más variables dependientes.. 7

sobreentrenamiento En aprendizaje automático se conoce como sobreentrenamiento el hecho de adaptar tanto el modelo a los datos de train, que lo hacemos demasiado específico para los mismos, dando malos resultados cuando lo evaluamos en los datos de test.. 16

verosimilitud marginal Representa la probabilidad a posteriori de haber obtenido los valores obtenidos en las observaciones en función de los parámetros de un kernel.. 12, 13, 17, 19, 20, 21, 23, 27

Acrónimos

CG Gradientes Conjugados. 24, 25

GP Proceso Gaussiano. 1, 2, 6, 7, 8, 9, 12, 15, 16, 17, 19, 23, 24, 27, 30, 31, 33

GPs Procesos Gaussianos. 1, 2, 3, 6, 15, 16, 17

MAP Maximum A Posteriori. 12

MVN Distribución normal multivariante. 6, 7

PCG Gradientes Conjugados Precondicionados. 25, 26, 27, 30

SE Square Exponential. 10, 12, 15, 16

SM Spectral Mixture. 15, 16, 17

SMP Spectral Mixture Product. 17, 29, 31

VA Variable aleatoria. 6

BIBLIOGRAFÍA

[1] R. P. A. Andrew Gordon Wilson. *Gaussian Process Kernels for Pattern Discovery and Extrapolation*. PhD thesis, Harvard University, 2013.

[2] K. E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, Universitty of Iowa, 1989.

[3] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Cambridge CB3 0FB, U.K., 2006.

[4] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. the MIT Press, Massachusetts Institute of Technology, 2012.

[5] K. B. Petersen and M. S. Pedersen. *The Matrix Cookbook*. ., http://matrixcookbook.com, 2012.

[6] C. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. the MIT Press, Massachusetts Institute of Technology, 2006.

[7] Y. Saatçi. *Scalable Inference for Structured Gaussian Process Models*. PhD thesis, University of Cambridge, 2011.

[8] J. L. Team. Julia lang official web, 2012. http://julialang.org/.

[9] J. L. Team. Optim.jl, basic api introduction, 2014. https://github.com/JuliaOpt/Optim.jl.

[10] A. G. Wilson. *Covariance Kernels for Fast Automatic Pattern Discovery and Extrapolation with Gaussian Processes*. PhD thesis, University of Cambridge, 2014.

[11] A. G. Wilson and E. Gilboa. *Fast Kernel Learning for Multidimensional Pattern Extrapolation*. PhD thesis, Carnegie Mellon University, 2014.

Código ejemplos SE

A

```
using Distances
using Gadfly
#Esta funcion devuelve la matriz de covarianzas que tiene en cada
#posicion la covarianza entre los dos elementos correspondientes
function kSE(X, l, w, sigma)
    dim = length(X)
    X = X/l;
    #Calculamos la matriz de distancias
    DIST = pairwise(Euclidean(), X)
    res = w.*exp(-0.5.*(DIST.^2)) + eye(dim).*sigma
end

function kSE_star(X, Xstar, l, w, sigma)
    dim_x = length(X)
    dim_xstar = length(Xstar)
    #Concatenamos los dos datos de train y los de test
    X = hcat(Xstar, X)
    #Reescalamos los sampleos con el parametro l
    X = X/l
    Xstar = Xstar/l
    DIST = pairwise(Euclidean(), X)
    K = w.*exp(-0.5.*(DIST.^2)) + eye(length(X)).*sigma
    #Solo me quedo con las covarianzas cruzadas
    res = K[1:dim_xstar, dim_xstar+1:(dim_xstar+dim_x)]
end

n_data = 6
x = [-5:10/n_data:5]
y = [3 -1 2 3 5 2 1]'
x_matrix = hcat(x)'
xstar = [-5:10/100:5]
xstar_matrix = hcat(xstar)'

w = 1
l = 1
sigma = 1e-5

Kxx = kSE(x_matrix, l,w,sigma)
Kxstarx = kSE_star(x_matrix, xstar_matrix, l, w, sigma)
```

```
40 Kxstarxstar = kSE(xstar_matrix, l, w, sigma)
41
42 m_posterior = Kxstarx*inv(Kxx)*y
43 v_posterior = diag(Kxstarxstar - Kxstarx * inv(Kxx) * Kxstarx')
44
45 lay1 = layer(x=x, y=y, Geom.point, Theme(default_color=color("blue")))
46 lay2 = layer(x=xstar, y = m_posterior,
47     Geom.line, Theme(default_color=color("red")))
48 lay3 = layer(x=xstar, y = m_posterior - sqrt(v_posterior), Geom.line,
49     Theme(default_color=color("green")))
50 lay4 = layer(x=xstar, y = m_posterior + sqrt(v_posterior), Geom.line,
51     Theme(default_color=color("green")))
52 myplot = Gadfly.plot(lay1, lay2, lay3, lay4,
53     Guide.manual_color_key("Leyenda",
54         ["Observaciones","Media", "Desviacion tipica"],
55         ["blue","red", "green"]))
```

CÓDIGO EJEMPLO SM

B

```
using Optim
using Distances
using Gadfly

function kSMP(X, phi, A)
    dim = length(X)
    DIST = pairwise(Euclidean(), X)
    sum = zeros(dim, dim)
    for i=1:3:A*3
    sum = sum + phi[i]^2.*exp(-2*pi^2*phi[i+1]^2.*(DIST.^2)).*cos(2*pi*phi
        [i+2].*DIST)
    end
    sum
end

function kSMP_star(X, Xstar, phi, A)
    dim_x = length(X)
    dim_xstar = length(Xstar)
    X = hcat(Xstar, X)
    DIST = pairwise(Euclidean(), X)
    sum = zeros(length(X), length(X))
    for i=1:3:A*3
    sum = sum + phi[i]^2.*exp(-2*pi^2*phi[i+1]^2.*(DIST.^2)).*cos(2*pi*phi
        [i+2].*DIST)
    end
    res = sum[1:dim_xstar, dim_xstar+1:(dim_xstar+dim_x)]
end

function marginal_likelihood(x, y, phi, A)
    n = length(x)
    K = kSMP(x, phi, A)
    cholK = chol(K)
    Kinv = inv(K)
    -n/2 * log(pi) - 0.5*sum(2.0*log(diag(cholK))) - 0.5*sum(y*Kinv*y')
end

function objetive(hyp::Vector)
    -marginal_likelihood(x, y, hyp, A)
end
```

```
38
39 x = hcat(collect(-8:16/100:8))'
40 y = sin(x) + 0.2*cos(x*0.5)
41
42 A = 10
43 phi = randn(A*3)
44
45 res_opt = optimize(objetive, method=:l_bfgs, phi, iterations=30,
       show_trace=true)
46 hyp_opt = res_opt.minimum
47
48 xstar = hcat(collect(-20:40/200:20))'
49 K = kSMP(x, hyp_opt, A)
50 Kxxstar = kSMP_star(x, xstar, hyp_opt, A)
51 mu_star = Kxxstar * inv(K) * y'
52
53 lay1 = layer(x=x, y=y, Geom.point, Theme(default_color=color("blue")))
54 lay2 = layer(x=xstar, y = mu_star, Geom.line, Theme(default_color=color("
       green")))
55 Gadfly.plot(lay1, lay2, Guide.manual_color_key("Leyenda", ["Datos de train
       ","Datos de test"], ["blue","green"]))
```